# INHERITANCE

# MODULE 5

## Content

- Inheritance
- Reusability in Object-Oriented Programming
- Redefining Members (Name Hiding)
- Overloading vs. Overriding
- Access Control
- Public and Private Inheritance
- Constructor, Destructor and Assignment Operator in Inheritance
- Multiple Inheritance
- Composition *vs* Inheritance

# Inheritance

- Inheritance is one of the ways in object-oriented programming that makes reusability possible.
- Reusability means taking an existing class and using it in a new programming situation.
- By reusing classes, you can reduce the time and effort needed to develop a program, and make software more robust and reliable.

# Inheritance

History

- The earliest approach to reusability was simply rewriting existing code. You have some code that works in an old program, but doesn't do quite what you want in a new project.
- You paste the old code into your new source file, make a few modifications to adapt it to the new environment. Now you must debug the code all over again. Often you're sorry you didn't just write new code.

## Inheritance

- To reduce the bugs introduced by modification of code, programmers attempted to create self-sufficient program elements in the form of functions.
- Function libraries were a step in the right direction, but, functions don't model the real world very well, because they don't include important data.
- All too often, functions require modification to work in a new environment.
- But again, the modifications introduce bugs.

## Reusability in OOP

- A powerful new approach to reusability appears in object-oriented programming is the class library. Because a class more closely models a real-world entity, it needs less modification than functions do to adapt it to a new situation.
- Once a class has been created and tested, it should (ideally) represent a useful unit of code.
- This code can be used in different ways again.

## Reusability in OOP

1. The simplest way to reuse a class is to just use an object of that class directly.
   - The standard library of the C++ has many useful classes and objects.
   - For example, **`cin`** and **`cout`** are such built in objects. Another useful class is **`string`** , which is used very often in C++ programs.

## Reusability in OOP

2. The second way to reuse a class is to place an object of that class inside a new class.
   - We call this “creating a member object.”
   - Your new class can be made up of any number and type of other objects, in any combination that you need to achieve the functionality desired in your new class.
   - Because you are composing a new class from existing classes, this concept is called composition (or more generally, aggregation). Composition is often referred to as a “has-a” relationship.

## Reusability in OOP

3. The third way to reuse a class is inheritance, which is described next. Inheritance is referred to as a "is a" or "a kind of" relationship.

## string

> While a character array can be fairly useful, it is quite limited. It's simply a group of characters in memory, but if you want to do anything with it you must manage all the little details.

> The Standard C++ string class is designed to take care of (and hide) all the low-level manipulations of character arrays that were previously required of the C programmer.

> To use strings you include the C++ header file **`<string>`**.

> Because of operator overloading, the syntax for using strings is quite intuitive (natural).

## string

```
#include <string>
#include <iostream>
using namespace std;
int main() {
    string s1, s2; // Empty strings
    string s3 = "Hello, World.";
    string s4("I am");
    s2 = "Today";
    s1 = s3 + " " + s4;
    s1 += " 20 ";
    cout << s1 + s2 + "!" << endl;
    return 0;
}
```

## string – Explanation (1 of 2)

- The first two strings, s1 and s2, start out empty, while s3 and s4 show two equivalent ways to initialize string objects from character arrays.
- You can assign to any string object using '**=**'. This replaces the previous contents of the string with whatever is on the right-hand side, and you don't have to worry about what happens to the previous contents – that's handled automatically for you.
- To combine strings you simply use the '**+**' operator, which also allows you to combine character arrays with strings.
- If you want to append either a string or a character array to another string, you can use the operator '**+=**'.

## string – Explanation (2 of 2)

> Finally, note that `cout` already knows what to do with strings, so you can just send a `string` (or an expression that produces a `string`, which happens with `s1 + s2 + "!"` directly to `cout` in order to print it.

## Inheritance

> OOP provides a way to modify a class without changing its code.

> This is achieved by using inheritance to derive a new class from the old one.

> The old class (called the base class) is not modified, but the new class (the derived class) can use all the features of the old one and additional features of its own.

## "*is a*" Relationship

> We know that PCs, Macintoshes and Cray are kinds of computers; a worker, a section manager and general manager are kinds of employee.

> If there is a "**kind of**" relation between two objects then we can derive one from other using the inheritance.

## Inheritance Syntax

> The simplest example of inheritance requires two classes: **a base class** and **a derived class**.

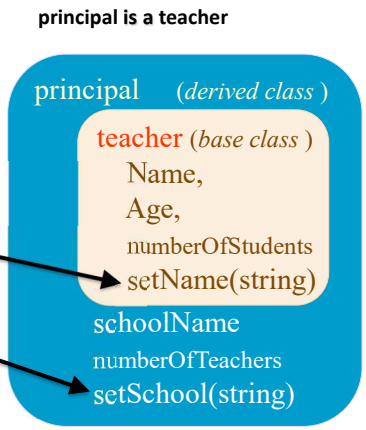
> The base class does not need any special syntax. The derived class, on the other hand, must indicate that it's derived from the base class.

> This is done by placing a colon after the name of the derived class, followed by a keyword such as public and then the base class name.

- Example: Modeling teachers and the principal (director) in a school.
- First, assume that we have a class to define teachers, then we can use this class to model the principal. Because the principal is a teacher.

```
class Teacher  {        // Base class
  private:          // means public for derived class members
     string name;
     int age, numberOfStudents;
   public:
     void setName (const string & new_name){ name = new_name; }
};
class Principal : public Teacher  { // Derived class
    string schoolName;              // Additional members
    int numberOfTeachers;
  public:
    void setSchool(const string & s_name){ schoolName = s_name; }
};
```

```
int main()  {
  Teacher t1;
  Principal p1;
  p1.setName(" Principal 1");
  t1.setName(" Teacher 1");
  p1.setSchool(" Elementary School");
  return 0;
}
```



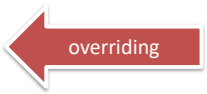
## Redefining Members (Name Hiding)

- Some members (data or function) of the base class may not suitable for the derived class. These members should be redefined in the derived class.
- For example, assume that the Teacher class has a print function that prints properties of teachers on the screen.
- But this function is not sufficient for the class Principal, because principals have more properties to be printed. So the print function must be redefined.

## Redefining Members

```
class Teacher {   // Base class
  protected:
    string name;
    int age, numOfStudents;
  public:
    void setName (const string & new_name) {
        name = new_name;
    }
    void print() const;
};
void Teacher::print() const  {
   cout << "Name: " << name<< "  Age: "
        << age << endl;
   cout << "Number of Students: "
        << numOfStudents << endl;
}
```

```
class Principal : public Teacher {
     string school_name;
     int numOfTeachers;
   public:
     void setSchool(const string & s_name) {
        school_name = s_name;
     }
     void print() const;
};
void Principal::print() const {
    cout << "Name: " << name << "  Age: "
         << age << endl;
    cout << "Number of Students: "
         << numOfStudents << endl;
    cout << "Name of the school: "
         <<  school_name << endl;
}
```




## Redefining Members

> Now the Principal class has two print() functions.

> The members of the base class can be accessed by using the scope operator (::).

```
void Principal::print() const    {
  Teacher::print();
  cout << "Name of the school: "
       << school_name << endl;
}
```

## Overloading *vs.* Overriding

- If you modify the signature and/or the return type of a member function from the base class then the derived class has two member functions with the same name.
- But this is not overloading, it is overriding.
- If the author of the derived class redefines a member function, it means he or she changes the interface of the base class.
- In this case the member function of the base class is hidden.

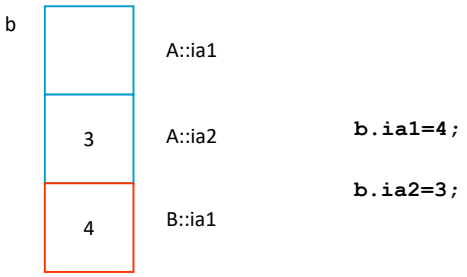
**Example**

```
class A {
 public:
   int ia1,ia2;
   void fa1();
   int fa2(int);
};
```

```
class B: public A {
 public:
   float ia1;
   float fa1(float);
};
```

```
int main(){
 B b;
 int j= b.fa2(1);
 b.ia1=4;
 b.ia2=3;
 float y= b.fa1(3.14);
 b.fa1();
 b.A::fa1();
 b.A::ia1=1;
}
```

# Example




# Access Control

> Remember, when inheritance is not involved, class member functions have access to anything in the class, whether public or private, but objects of that class have access only to public members.

> Once inheritance enters the picture, other access possibilities arise for derived classes. Member functions of a derived class can access public and protected members of the base class, but not private members. Objects of a derived class can access only public members of the base class.

| Access | Base Class | Derived Class | Object |
|---|---|---|---|
| public | yes | yes | yes |
| protected | yes | yes | no |
| private | yes | no | no |

```
class A {
 private:
   int ia1;
 protected:
   int ia2;
 public:
   void fa1();
   int fa2(int);
};
```

```
class B: public A {
 private:
  float ia1;
 public:
  float fa1(float);
};
```

```
float B::fa1(float f){
    A::ia1= 2.22 ;
    ia2=static_cast<int>(f*f);
}
```

```
class Teacher {                  // Base class
  private:                       // only members of Teacher can access
    string name;
  protected:                     // Also members of derived classes can
    int age, numOfStudents;
  public:                        // Everyone can access
    void setName (const string & new_name){ name = new_name; }
    void print() const;
};
class Principal : public Teacher {        // Derived class
  private:                                // Default
    string  school_name;
    int numOfTeachers;
  public:
    void setSchool(const string & s_name) { school_name = s_name; }
    void print() const;
    int getAge() const { return age; }  // it works because age is protected
    const string & get_name(){ return name;} // ERROR! name is private
};
```

```
int main()
{
    teacher t1;
    principal p1;
    t1.numberOfStudents=54;
    t1.setName("Sevda Yaman");
    p1.setSchool("Halide Edip Adivar Lisesi");
 }
```

## Protected vs. Private Members

- In general, class data should be private. Public data is open to modification by any function anywhere in the program and should almost always be avoided.
- Protected data is open to modification by functions in any derived class. Anyone can derive one class from another and thus gain access to the base class's protected data. It's safer and more reliable if derived classes can't access base class data directly.
- But in real-time systems, where speed is important, function calls to access private members is a time-consuming process. In such systems data may be defined as protected to make derived classes access data directly and faster.

## Private data: Slow and reliable

```
class A { // Base class
  private:
    int i;  // safe
  public:
    void access(int new_i){
        if (new_i > 0 && new_i <= 100) i=new_i;
    }
};
class B : public A { // Derived class
    int k;
  public:
    void set(int new_i, int new_k){
       A::access(new_i);  // reliable but slow
       :
    }
};
```

## Protected data: Fast, author of the derived class is responsible

```
class A {       // Base class
  protected:
    int i;  // derived class can access directly
 public:
    :
};
class B : public A { // Derived class
    int k;
  public:
    void set(int new_i,int new_k){
       i= new_i; // fast
       :
    }
};
```

## public Inheritance

- In inheritance, you usually want to make the access specifier public.

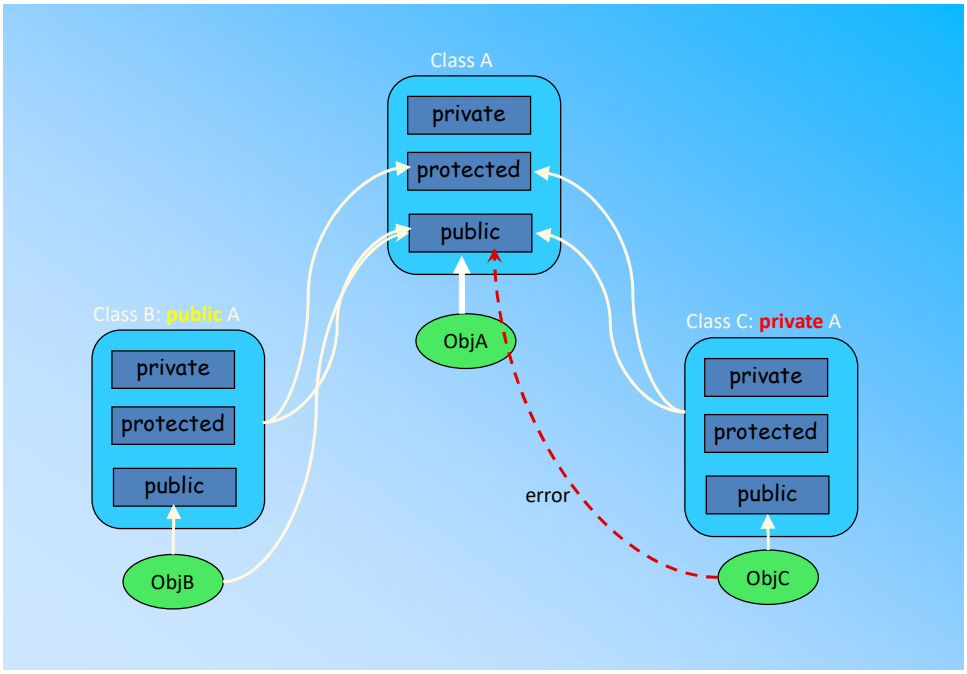
```
class Base {   };
class Derived : public Base { }
```

- This is called public inheritance (or sometimes public derivation). The access rights of the members of the base class are not changed.
- Objects of the derived class can access public members of the base class.
- Public members of the base class are also public members of the derived class.

## private Inheritance

```
class Base
    {   };
class Derived : private Base { }
```

- This is called private inheritance.
- Now public members of the base class are private members of the derived class.
- Objects of the derived class can not access members of the base class.
- Member functions of the derived class can still access public and protected members of the base class.




## Redefining Access

> Access specifications of public members of the base class can be redefined in the derived class.

> When you inherit privately, all the public members of the base class become private.

> If you want any of them to be visible, just say their names (no arguments or return values) along with the using keyword in the public section of the derived class:

```
class Base {
  private:
   int k;
  public:
   int i;
   void f();
};
```

```
class Derived : private Base
{
   int m;
 public:
   // f() is public again
   Base::f();
   void fb1();
};
```

```
int main(){
   Base b;
   Derived d;
   b.i=5;     // OK
   d.i=0;  // ERROR !
   b.f();      // OK
   d.f();      // OK
   return 0;
};
```

```
class Base {
  private:
   int k;
  public:
   int i;
   void f();
};
```

```
class Derived : public Base {
  int m;
Base::i; // i is private

public:
  void fb1();
};
```

```
int main(){
   Base b;
   Derived d;
   b.i=5;     // OK
   d.i=0;  // ERROR !
   b.f();      // OK
   d.f();      // OK
   return 0;
};
```

```
class Base {
  private:
    int k;
  public:
    int i;
    void f();
    void f(int);
    bool f(int,float);
};
```

```
class Derived : private Base {
    int m;
 public:
    Base::f(int);
    void fb1();
};
```

```
int main(){
    Base b;
    Derived d;
    b.i=5;     // OK
    d.i=0;     // ERROR
    b.f();     // OK
    d.f(42);   // OK
    d.f(108,3.14) // ERROR
    return 0;
};
```

## Special Member Functions and Inheritance

- Some functions will need to do different things in the base class and in the derived class:
  - Overloaded `operator=`,
  - The destructor,
  - All constructors.
- Consider a constructor.
  - The base class constructor must create the base class data, derived class constructor must create the derived class data.
  - Because the derived class and base class constructors create different data, one constructor cannot be used in place of another.
  - Constructor of the base class can not be the constructor of the derived class.

## Special Member Functions and Inheritance

- Similarly, the `operator=` in the derived class must assign values to derived class data, and the `operator=` in the base class must assign values to base class data.
- These are different jobs, so assignment operator of the base class can not be the assignment operator of the derived class.

## Constructors and Inheritance

- When you define an object of a derived class, the base class constructor will be called before the derived class constructor.
- This is because the base class object is a sub-object—a part—of the derived class object, and you need to construct the parts before you can construct the whole.
- If the base class has a constructor that needs arguments, this constructor must be called before the constructor of the derived class.

```
class Teacher {
    char *Name;
    int age,numberOfStudents;
  public:
    Teacher(char *newName){Name=newName;}
};
class Principal : public Teacher {
    int numberOfTeachers;
  public:
    Principal(char *, int );
};
```

```
  Principal::Principal(const string& new_name,
              int numOT):Teacher(new_name) {
       numOfTeachers = numOT;
  }
```

> Remember, the constructor initializer can also be used to initialize members.

```
  Principal::Principal(const string& new_name,
            int numOT):Teacher(new_name),
                       numOfTeachers(numOT){ }
  int main()  {
      Principal p1("Ali Bilir", 20);
      return 0;
  }
```

## Constructors and Inheritance

> If the base class has a constructor, which must take some arguments, then the derived class must also have a constructor that calls the constructor of the base with proper arguments.

## Destructors and Inheritance

> Destructors are called automatically.

> When an object of the derived class goes out of scope, the destructors are called in reverse order: The derived object is destroyed first, then the base class object.

```
#include <iostream>
using namespace std;
class B {
   public:
      B() { cout << "B constructor" << endl; }
     ~B() { cout << "B destructor" << endl; }
};
class C : public B {
  public:
     C() { cout << "C constructor" << endl; }
     ~C() { cout << "C destructor" << endl; }
};
int main(){
   cout << "Start" << std::endl;
   C c;  // create a C object
   cout << "End" << std::endl;
}
```

```
#include <iostream>
using namespace std;
class B {
   public:
      B() { cout << "B constructor" << endl; }
     ~B() { cout << "B destructor" << endl; }
};
class C : public B {
  public:
     C(): B() { cout << "C constructor" << endl; }
     ~C() { cout << "C destructor" << endl; }
};
int main(){
   cout << "Start" << std::endl;
   C c;  // create a C object
   cout << "End" << std::endl;
}
```

```
#include <iostream>
using namespace std;
class B {
   public:
      B(int i) { cout << "B constructor" << endl; }
     ~B() { cout << "B destructor" << endl; }
};
class C : public B {
  public:
     C() { cout << "C constructor" << endl; }
     ~C() { cout << "C destructor" << endl; }
};
int main(){
   cout << "Start" << std::endl;
   C c;  // create a C object
   cout << "End" << std::endl;
}
```

```
#include <iostream>
using namespace std;
class B {
   public:
      B(int i) { cout << "B constructor" << endl; }
     ~B() { cout << "B destructor" << endl; }
};
class C : public B {
  public:
     C():B(42) { cout << "C constructor" << endl; }
     ~C() { cout << "C destructor" << endl; }
};
int main(){
   cout << "Start" << std::endl;
   C c;  // create a C object
   cout << "End" << std::endl;
}
```

## Example: Constructor Chain

```
#include <iostream>
using namespace std;
class A {
   private:
     int x;
     float y;
   public:
     A(int i, float f) :
        x(i), y(f) { // initialize A
         cout << "Constructor A" << endl;
     }
     void display() {
         cout << x << ", " << y  << endl;
     }
};
```

## Example: Constructor Chain

```
class B : public A {
   private:
     int v;
     float w;
   public:
      B(int i1, float f1, int i2, float f2) :
         A(i1, f1),  // initialize A
         v(i2), w(f2) { // initialize B
             cout << "Constructor B" << endl;
      }
      void display(){
         A::display();
         cout << v << ", " << w << endl;
      }
};
```

## Example: Constructor Chain

```
class C : public B {
 private:
   int r;
   float s;
 public:
 C(int i1,float f1,int i2,float f2,int i3,float f3):
         B(i1, f1, i2, f2), // initialize B
         r(i3), s(f3) {     // initialize C
      cout << "Constructor C" << endl;
   }
   void display() {
      B::display();
      cout << r << ", " << s << endl;
   }
};
```

## Example: Constructor Chain

```
int main() {
   C c(1, 1.1, 2, 2.2, 3, 3.3);
   cout << "\nData in c = ";
   c.display();
}
```

## Explanation

- **C** class is inherited from a **B** class, which is in turn inherited from **A** class.
- Each class has one **`int`** and one **`float`** data item.
- The constructor in each class takes enough arguments to initialize the data for the class and all ancestor classes. This means ***two*** arguments for the **A** class constructor, ***four*** for **B** (which must initialize **A** as well as itself), and ***six*** for **C** (which must initialize **A** and **B** as well as itself).
- Each constructor calls the constructor of its base class.

## Explanation

- In **`main()`**, we create an object of type **C**, initialize it to ***six*** values, and display it.
- When a constructor starts to execute, it is guaranteed that all the sub-objects are created and initialized.
- Incidentally, you can't skip a generation when you call an ancestor constructor in an initialization list. In the following modification of the C constructor:

```
C(int i1,float f1,int i2,float f2,int i3,float f3) : A(i1, f1),  //
ERROR! can't initialize A
 intC(i3), floC(f3)        // initialize C
{  }
```

- the call to **`A()`** is illegal because the A class is not the immediate base class of **C**.

## Explanation: Constructor Chain

- You never need to make explicit destructor calls because there's only one destructor for any class, and it doesn't take any arguments.
- The compiler ensures that all destructors are called, and that means all of the destructors in the entire hierarchy, starting with the most-derived destructor and working back to the root.

## Assignment Operator and Inheritance

- Assignment operator of the base class can not be the assignment operator of the derived class.
- Recall the **`String`** example.

```
class String {
 protected:
   int size;
   char *contents;
 public:
   const String & operator=(const String&);
   : // Other methods
};
```

## Assignment Operator and Inheritance

```
const String& String::operator=(
                        const String &in)   {
    size = in.size;
    delete[] contents;
    contents = new char[size+1];
    strcpy(contents, in.contents);
    return *this;
}
```

## Example

> Class **String2** is derived from class **String**. If an assignment operator is necessary it must be written

```
class String2 : public String {
  int size2;
  char *contents2;
 public:
  const String2& operator=(const String2&);
  :
};
```

## Example

```
const String2& String2::operator=(const String2 &in)  {
    size = in.size;
    delete[] contents;
    contents= strdup(in.contents);
    size2 = in.size2;
    delete[] contents2;
    contents2 = strdup(in.contents2);
    return *this;
}
```

- In previous example, data members of **String** (Base) class must be protected. Otherwise methods of the **String2** (Derived) can not access them.
- The better way to write the assignment operator of **String2** is to call the assignment operator of the **String** (Base) class.
- Data members of **String** (Base) class may be private.

```
const String2& String2::operator=(const String2& in){
  String::operator=(in);
  cout << "Assignment operator of String2 is invoked";
  size2 = in.size2;
  delete[] contents2;
  contents2 = new char[size2 + 1];
  strcpy(contents2, in.contents2);
  return *this;
}
```

## String::operator=(in_object);

- In this method the assignment operator of the String is called with an argument of type (**`String2 &`**).
  - Actually, the operator of String class expects a parameter of type (**`String &`**).
- This does not cause a compiler error, because as we will see in Module 7, a reference to base class can carry the address of an object of derived class.

## Composition *vs*. Inheritance

- Every time you place instance data in a class, you are creating a "has a" relationship.
- If there is a class **`Teacher`** and one of the data items in this class is the teacher's name, I can say that a **`Teacher`** object has a name.
- This sort of relationship is called ***composition*** because the **`Teacher`** object is composed of these other variables.
- Remember the class **`ComplexFrac`**. This class is composed of two **`Fraction`** objects.
- Composition in OOP models the real-world situation in which objects are composed of other objects.

# Composition *vs*. Inheritance

> Inheritance in OOP mirrors the concept that we call generalization in the real world. If I model workers, managers and researchers in a factory, I can say that these are all specific types of a more general concept called an employee.

> Every kind of employee has certain features: name, age, IDnum, and so on.

> But a manager, in addition to these general features, has a department that he/she manages.

> A researcher has an area on which he/she studies.

> In this example the manager has not an employee.

> The manager is an employee

> You can use composition & inheritance together.

> The following example shows the creation of a more complex class using both of them.

```
class A {
  int i;
public:
  A(int ii):i(ii){}
  ~A() {}
  void f() const {}
};
```

```
class B {
  int i;
public:
 B(int ii):i(ii) {}
  ~B() {}
  void f() const {}
};
```

```
class C : public B {
  A a;
public:
  C(int ii): B(ii), a(ii) {}
  ~C() {}
  void f() const {
    a.f();
    B::f();
  }
};
```

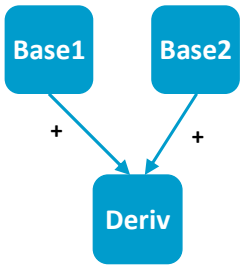
- C inherits from B and has a member object ("is composed of") of type A.
- You can see the constructor initializer list contains calls to both the base-class constructor and the member-object constructor.
- The function C::f( ) redefines B::f( ), which it inherits, and also calls the base-class version. In addition, it calls a.f( ).
- Notice that the only time you can talk about redefinition of functions is during inheritance; with a member object you can only manipulate the public interface of the object, not redefine it.
- In addition, calling f( ) for an object of class C would not call a.f( ) if C::f( ) had not been defined, whereas it would call B::f( ).

# Multiple Inheritance

```
class Base1{
  public:
   int a;
   void fa1();
   char *fa2(int);
};
```

```
class Base2{
 public:
  int a;
  char *fa2(int,char*);
  int fc();
};
```

```
class Deriv:public Base1,public Base2
{
public:
    int a;
    float fa1(float);
    int fb1(int);
};
```




```
int main(){
   Deriv d;
   d.a=4;
   float y=d.fa1(3.14);
   int i=d.fc();
}
```

```
char* c=d.fa2(1);
```

**is not valid**.

In inheritance functions are not overloaded.

They are **overridden**.

You have to write

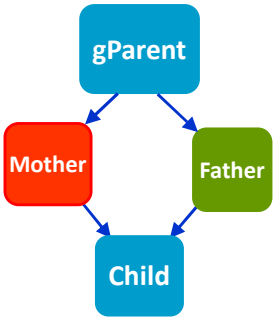
```
char* c = d.Base1::fa2(1);
```

or

```
char * c=
  d.Base2::fa2(1,"Hello");
```

## Repeated Base Classes

```
struct gParent{
  int x;
};
class Mother : public gParent{
};
class Father : public gParent{
};
struct Child : public Mother,public Father{
    void fun(){
      x = 42; ✗
    }
};
```




## Repeated Base Classes

- Both `Mother` and `Father` inherit from `gParent`, and `Child` inherits from both `Mother` and `Father`.
- Recall that each object created through inheritance contains a sub-object of the base class.
- A `Mother` object and a `Father` object will contain sub-objects of `gParent`, and a `Child` object will contain sub-objects of `Mother` and `Father`, so a `Child` object will also contain two `gParent` sub-objects, one inherited via `Mother` and one inherited via `Father`.
- This is a strange situation.
  - There are two sub-objects when really there should be one.

## Repeated Base Classes

```
class gParent {
   protected:
      int gdata;
 };
class Child : public Mother, public Father {
      void Cfunc() {
         int temp = gdata;  // error: ambiguous
      }
 };
```

> The compiler will complain that the reference to **gdata** is ambiguous. It doesn't know which version of **gdata** to access:
  – the one in the **gParent** subobject in the **Mother**
  – the one in the **gParent** subobject in the **Father** subobject.

## Solution: Virtual Base Classes

> You can fix this using a new keyword, **virtual**, when deriving **Mother** and **Father** from **gParent**

```
class gParent { };
class Mother : virtual public gParent { };
class Father : virtual public gParent { };
class Child : public Mother,public Father {
};
```

## Solution: Virtual Base Classes

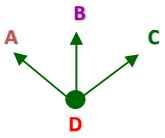
> **`virtual`** keyword tells the compiler to inherit only one subobject from a class into subsequent derived classes.

> That fixes the ambiguity problem, but other more complicated problems arise that are too complex to delve into here.

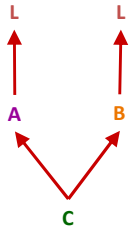
> In general, you should avoid multiple inheritance, although if you have considerable experience in C++, you might find reasons to use it in unusual situations.

```
class Base
{
  public:
    int a,b,c;
};
class Derived : public Base
{
  public:
    int b;
};
class Derived2 : public Derived
{
  public:
    int c;
};
```

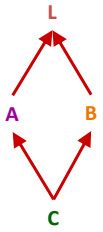
```
class A {
   ...
};
class B {
   ...
};
class C {
   ...
 };
class D : public A, public B, private C {
  ...
 };
```



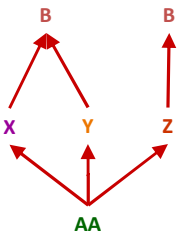

```
class L {
   public:
     int next;
};
class A : public L {
   ...
};
class B : public L {
   ...
};
class C : public A, public B {
  void f() ;
  ...
 };
```



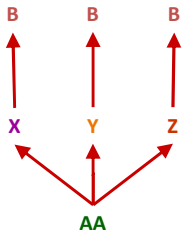
```
class L {
   public:
     int next;
};
class A : virtual public L {
   ...
};
class B : virtual public L {
   ...
};
class C : public A, public B {
  ...
 };
```



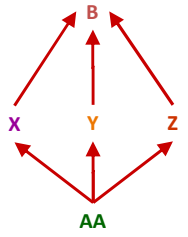

```
class B {
    ...
};
class X : virtual public B {
   ...
};
class Y : virtual public B {
   ...
};
class Z : public B {
  ...
 };
class AA : public X, public Y , public Z {
  ...
 };
```



```
class B {
    ...
};
class X : virtual public B {
   ...
};
class Y : public B {
   ...
};
class Z : public B {
  ...
 };
class AA : public X, public Y , public Z {
  ...
 };
```




```
class B {
    ...
};
class X : virtual public B {
   ...
};
class Y : virtual public B {
   ...
};
class Z : virtual public B {
  ...
 };
class AA : public X, public Y , public Z {
  ...
 };
```



# Implementation of Multiple & Virtual Inheritance

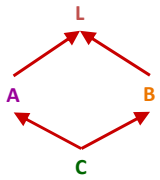
> Compiler inserts a pointer to virtual table for each virtual base classes to manage the multiple inheritance

> We will study virtual tables in Module 7.

> But for now, consider the following question:

– C obj;

• sizeof(obj) ?




```
class L {
   int top;
};
class A : virtual public L {
    int a;
};
class B : virtual public L {
     int b;
};
class C : public A, public B {
     int c;
};
```